Final Exam

- Thursday, December 18th
- 10:25am – 12:25pm
- Location: Science II 140
- One page of letter-size, one-sided summary sheet allowed.
- No cellphone, smartphone, tablet, laptop, etc.
- Be on time!
Disclaimer

• This review helps you better prepare for the final exam.
• It is by no means comprehensive.
• Study all materials for the exam.
Topics before midterm

• Introduction to Distributed Systems
• Distributed Architectures
• Communication in Distributed Systems
• RPC & RMI
• Naming
• Synchronization & Coordination
• Client and Server Design Issues
Topics after midterm

- Consistency models
  - data-centric, client-centric
- Implementing consistency models
- Transactions
  - Serializability
  - Concurrency control: locking, opportunistic concurrency control, timestamp based concurrency control
- Atomic commit protocols
  - 2-Phase, 3-Phase
- CAP theorem
- Fault tolerance
- Case studies: Amazon Dynamo, Google Chubby, BigTable
Mechanisms for sequential consistency

• Primary-based replication protocols
• Replicated-write protocols
  • Active replication using multicast communication
  • Quorum-based protocols
The passive (primary-backup) model

Front ends only communicate with primary
Active replication using multicast

• Active replication
  • Front end multicasts request to each replica using a *totally ordered reliable multicast*
  • System achieves sequential consistency but not linearizability
    • Total order in which replica managers process requests may not be same as real-time order in which clients made requests
Active replication
Implementing ordered multicast

- Incoming messages are held back in a queue until delivery guarantees can be met
- Coordination between all machines needed to determine delivery order
- FIFO-ordering
  - easy, use a separate sequence number for each process
- Total ordering (two implementation approaches)
  - Use a sequencer
  - Distributed algorithm with three phases
- Causal ordering
  - use vector timestamps
Quorum-based protocols

- Assign a number of votes to each replica
- Let $N$ be the total number of votes
- Define $R =$ read quorum, $W =$ write quorum
- $R + W > N$
- $W > N/2$
- Only one writer at a time can achieve write quorum
- Every reader sees at least one copy of the most recent read (takes one with most recent version number)
Quorum-based protocols

Three examples of the voting algorithm:

a) A correct choice of read and write set
b) A choice that may lead to write-write conflicts
c) A correct choice, known as ROWA (read one, write all)
Possible policies

- **ROWA**: $R=1, W=N$
  - Fast reads, slow writes (and easily blocked)
- **RAWO**: $R=N, W=1$
  - Fast writes, slow reads (and easily blocked)
- **Majority**: $R=W=N/2+1$
  - Both moderately slow, but extremely high availability
- **Weighted voting**
  - Give more votes to “better” replicas
Concurrency control

- Motivation: without concurrency control, we have lost updates, inconsistent retrievals, dirty reads, etc.
- Concurrency control schemes are designed to allow two or more transactions to be executed correctly while maintaining serial equivalence
  - Serial Equivalence is correctness criterion
    - Schedule produced by concurrency control scheme should be equivalent to a serial schedule in which transactions are executed one after the other
- Schemes: locking, optimistic concurrency control, timestamp based concurrency control
## Lock compatibility

<table>
<thead>
<tr>
<th>For one object</th>
<th>Lock already set</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>none</td>
<td>read</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>write</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>write</td>
<td>wait</td>
</tr>
</tbody>
</table>
Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)

2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.
Strict Two-Phase Locking

Growing phase

Lock point

Shrinking phase

All locks are released at the same time
## Deadlock with write locks

<table>
<thead>
<tr>
<th>Transaction $T$</th>
<th>Operations</th>
<th>Locks</th>
<th>Transaction $U$</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td>$a.deposit(100)$;</td>
<td>write lock $A$</td>
<td>Operations</td>
<td>$b.deposit(200)$;</td>
<td>write lock $B$</td>
</tr>
<tr>
<td></td>
<td>$b.withdraw(100)$</td>
<td>waits for $U$'s</td>
<td></td>
<td>$a.withdraw(200)$;</td>
<td>waits for $T$'s</td>
</tr>
<tr>
<td></td>
<td>⋮</td>
<td>lock on $B$</td>
<td></td>
<td>⋮</td>
<td>lock on $A$</td>
</tr>
</tbody>
</table>
Optimistic concurrency control

- **Drawbacks of locking**
  - Overhead of lock maintenance
  - Deadlocks
  - Reduced concurrency
- **Optimistic Concurrency Control**
  - In most applications, likelihood of conflicting accesses by concurrent transactions is low
  - Transactions proceed as though there are no conflicts
  - Three phases
    - Working Phase – transactions read and write private copies of objects
    - Validation Phase – each transaction is assigned a transaction number when it enters this phase
    - Update Phase
Optimistic concurrency control: serializability of transaction $T_v$ w.r.t. $T_i$

$T_v$ and $T_i$ are overlapping transactions

For $T_v$ to be serializable w.r.t $T_i$ the following rules must hold

<table>
<thead>
<tr>
<th>$T_v$</th>
<th>$T_i$</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>read</td>
<td>1. $T_i$ must not read objects written by $T_v$</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
<td>2. $T_v$ must not read objects written by $T_i$</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>3. $T_i$ must not write objects written by $T_v$ and $T_v$ must not write objects written by $T_i$</td>
</tr>
</tbody>
</table>

If simplification is made that only one transaction may be in its validation or write phases at one time, then third rule is always satisfied
Validation of transactions

Backward validation of transaction $T_v$

boolean valid = true;
for (int $T_i = startTn+1; T_i <= finishTn; T_i++){
    if (read set of $T_v$ intersects write set of $T_i$) valid = false;
}

Forward validation of transaction $T_v$

boolean valid = true;
for (int $T_id = active1; T_id <= activeN; T_id++){
    if (write set of $T_v$ intersects read set of $T_id$) valid = false;
}
Timestamp based concurrency control

- Each transaction is assigned a unique timestamp at the moment it starts
  - In distributed transactions, Lamport’s timestamps can be used
- Every data item has a timestamp
  - Read timestamp = timestamp of transaction that last read the item
  - Write timestamp = timestamp of transaction that most recently changed an item
Operation conflicts for timestamp ordering

<table>
<thead>
<tr>
<th>Rule</th>
<th>$T_c$</th>
<th>$T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>write</td>
<td>read</td>
</tr>
<tr>
<td></td>
<td>$T_c$ must not write an object that has been read by any $T_i$ where $T_i &gt; T_c$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>this requires that $T_c \geq$ the maximum read timestamp of the object.</td>
<td></td>
</tr>
</tbody>
</table>

| 2.   | write | write |
|      | $T_c$ must not write an object that has been written by any $T_i$ where $T_i > T_c$ |
|      | this requires that $T_c >$ write timestamp of the committed object. |

| 3.   | read  | write |
|      | $T_c$ must not read an object that has been written by any $T_i$ where $T_i > T_c$ |
|      | this requires that $T_c >$ write timestamp of the committed object. |
Timestamp ordering write rule

if ($T_c \geq$ maximum read timestamp on $D$ && $T_c \gt$ write timestamp on committed version of $D$)
   perform write operation on tentative version of $D$ with write timestamp $T_c$
else /* write is too late */
   Abort transaction $T_c$
Timestamp ordering read rule

if ( $T_c > \text{write timestamp on committed version of } D$) {
    \text{let } D_{\text{selected}} \text{ be the version of } D \text{ with the maximum write timestamp } \leq T_c \\
    \text{if (} D_{\text{selected}} \text{ is committed)} \\
    \quad \text{perform read operation on the version } D_{\text{selected}} \\
    \text{else} \\
    \quad \text{Wait until the transaction that made version } D_{\text{selected}} \text{ commits or aborts} \\
    \quad \text{then reapply the read rule} \\
} \text{else} \\
\text{Abort transaction } T_c
Two-Phase Commit

(a) The finite state machine for the coordinator in 2PC.
(b) The finite state machine for a participant.
Three-Phase Commit

a) Finite state machine for the coordinator in 3PC
b) Finite state machine for a participant
What goals might you want from a shared-data system?

- **Strong Consistency**: all clients see the same view, even in the presence of updates

- **High Availability**: all clients can find some replica of the data, even in the presence of failures

- **Partition-tolerance**: the system properties hold even when the system is partitioned
CAP conjecture (Brewer)

• You can only have two out of these three properties

• The choice of which feature to discard determines the nature of your system
Fault tolerance of process groups

- A system is $k$ fault tolerant if it can survive faults in $k$ components and still meets its specification.
- If failures are safe (silent), then $k+1$ processes are sufficient to get $k$ fault tolerance.
- In case of arbitrary failures, $2k+1$ processes are required (since $k$ failing processes can all generate the same result by chance)
Byzantine agreement result

• In a system with $k$ faulty processes agreement can be achieved only if there are $2k+1$ functioning correctly
Good luck!

- Thanks for joining us for the class.
- Wish you a fruitful final season and happy holidays!